Beyond Instruction Level Parallelism - Multithreading

Lecture #12: Monday, May 8, 2000
Lecturer: Professor Bill Dally
Scribe: Melvyn Lim

1 *Introduction*

The main idea behind parallel execution is to extract parallelism from the inherently sequential nature of programs. A program can be thought of as a series of instructions executed in a logical order over time. Where later instructions do not depend on earlier ones, there is Instruction Level Parallelism (ILP). In Figure 1, each arrow represents a basic block, a subroutine, or any logical subset of instructions ending with a control transfer instruction. The series of adjacent arrows represents the sequence of instructions actually executed as the program runs.

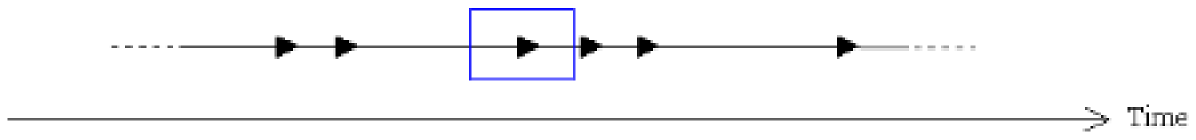

Figure 1

The goal in ILP is to find instructions that are not data dependent so that they can be executed concurrently to achieve higher instruction throughput. The standard way to do this is to look for data independent instructions within a window of instructions, usually of a fixed size. In Figure 1, the box represents an instruction window of, say, 16 instructions. The processor fetches 16 instructions to fill the window and tries to find independent instructions (within the window) to issue in the same cycle.

To find even more parallelism, a natural extension is to make the instruction window bigger. Examining more instructions together increases the chance of finding independent instructions, although there are diminishing returns. However, the logic necessary to check for independence becomes forbiddingly complex and makes large instruction windows impractical.

Parallelism can be extracted between different threads too. Consider 2 independent threads as shown in Figure 2. The same technique of finding independent instructions within a window can be applied to each of these threads. The resulting independent instructions from both threads can then be issued either to separate execution pipelines or to any execution pipeline available.

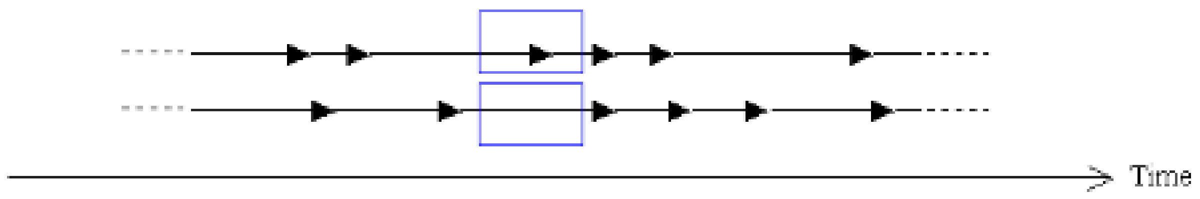

Figure 2

One recurring theme in ILP is that data dependencies occur very frequently and limit the amount of parallelism that can be extracted, and hence the realizable performance improvement. To alleviate this problem, speculation is applied. The easy approach is to assume that instructions are independent and issue them, then check later if the assumption was correct. If the instructions were not independent, then the dependent instruction and the instructions following it have to be re-issued. Speculation is usually applied to dependencies through memory only, since other dependencies can be tracked through registers. Consider the example of a data dependency through memory in Figure 3.

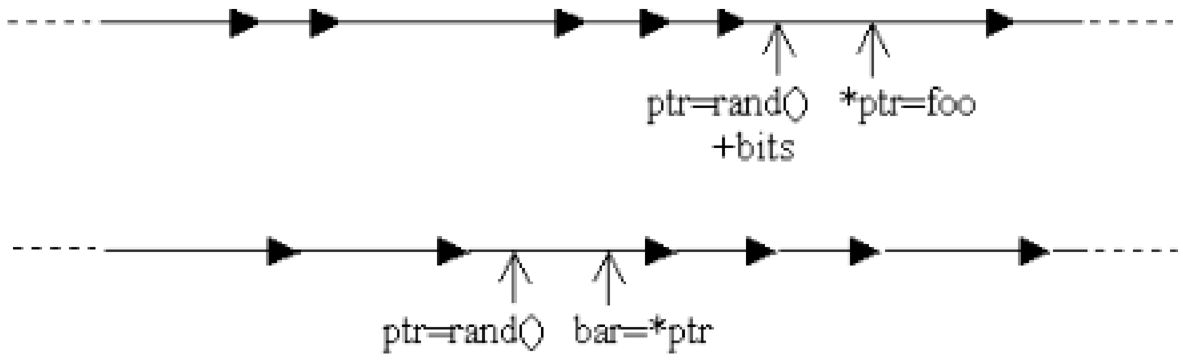

Figure 3

Assume that the 4 statements have instructions occurring in the same window. Strictly speaking, it is uncertain whether ptr will have different values in the 2 threads, hence the ptr=foo and bar=ptr statements cannot be independently issued. However, using speculation, ptr can be assumed to have different values in the 2 threads and the *ptr=foo and bar=*ptr statements can be independently issued. If ptr is found to be equal in the 2 threads, then the statements have to be re-issued.

2 *Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results (Weber, Gupta)*

This paper discusses running multiple contexts (programs, threads, etc.) per processor on a multiprocessor. Each processor contains a small fixed number of contexts with independent register sets to enable short context switch times.

3 levels of context switching times are mentioned: 1 cycle, if multiple copies of the pipeline registers are kept; 4 cycles, if the pipeline has to be drained and filled; and 16 cycles, if registers are loaded from and stored to some local memory. The criterion for context switching is a cache miss or a write hit to read-shared data.

The figure of merit used in this paper is processor efficiency, which was defined as the number of cycles spent doing useful work over the total number of cycles. However, it was noted that the total number of cycles considered probably did not include the context switch overhead. Also, the total number of cycles was counted only for as long as all traces used were running. As soon as a trace completed on each processor, statistics were no longer taken. Thus, idle cycles (spent waiting for the next thread to execute) are not taken into account. This is a counterintuitive measure of processor efficiency, since a large part of the inefficiency of multiprocessors comes from the inactivity of the individual processors after completing a thread. As a result, the processor efficiency reported was probably significantly higher than it should be.

The trace simulation results reported run length (the number of cycles between context switches) and read and write latencies (the number of cycles needed to complete reads and writes to service the cache miss). An interesting point is the ratio of the run length to the read latency. In Table 2 of the paper, MP3D has a run length of 16 cycles and a read latency of 32 cycles. This means that after a cache miss 2 context switches have to occur before the read servicing the cache miss returns, implying that several traces/processes (contexts) must be active in order to keep the processor busy. Given the larger read and write latencies of processors today, this effect would be even more pronounced. Keeping the processor busy would require a large number of actively running processes, which many applications might not be able to support.

The graphs in Figures 2 through 10 plot processor efficiency against network delay running each of the 3 benchmarks for different numbers of contexts and different context switch overhead. Network delay is the time taken for signals to traverse the interconnection network. It was noted that in Figure 5 of the paper, P-Thor peaked at about 89% even for a context switch overhead of only 1 cycle. This was probably due to the limited amount of ILP in the program.

The use of a single cache for multiple contexts leads to both positive and negative cache interference. Positive interference occurs when the data used by one context is also used by another context. Negative interference occurs when the data used by one context knocks out the data used by another context. Negative interference tends to dominate. However, if each processor handled every nth iteration of a loop, much more positive interference will occur, since data tends to be reused in a loop.

The effectiveness of the multiple contexts in a multiprocessor depends on the magnitude of the context switch overhead. If it is high, then it might be better to just stall the processor on a cache miss.

3 *Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism (Keckler, Dally)*

This paper discusses processor coupling, a runtime scheduling mechanism in which multiple functional units execute instructions from multiple threads that directly access one another's register files. A hardware scheduling mechanism interleaves multiple threads among the multiple functional units to maintain high processor usage.

In the simulations conducted, 5 modes were used. In Sequential (SEQ) mode, only one thread is statically scheduled and executed on one pipeline. In Statically Scheduled mode (STS), one thread is statically scheduled and executed on multiple pipelines. In Ideal mode, one thread is statically scheduled and is extensively loop-unrolled. In Thread Per Element (TPE) mode, multiple threads are statically scheduled to be executed, each on its own pipeline. In Coupled mode, each thread is first statically scheduled to be executed on multiple pipelines (like in STS mode), then all the threads are dynamically scheduled to be executed on the same set of pipelines. This results in the individual thread schedules being interleaved, and some instructions are necessarily executed later than specified in the original static schedule due to finite execution resources.

The STS mode is most similar to the VLIW architecture; it does not rely on the presence of many active threads to sustain high processor usage, although its performance is limited by the amount of parallelism in a single thread. The TPE mode is most similar to the multiprocessor architecture; it requires multiple active threads to sustain high processor usage, although parallelism within individual threads is not essential to its performance.

Interference between multiple threads of the same program occurs because the different threads access the same variables. An experiment was conducted in the STS and Coupled modes to show this effect. For the Coupled mode, 4 threads of a benchmark were created and each thread accessed a common queue of devices to be evaluated, chose a device, updated the queue, and evaluated the device. There was only 1 thread for the STS mode. Table 3 of the paper shows the compile time schedule length and average runtime cycle count (to evaluate one device). Interference causes the runtime cycle count for all 4 threads in the Coupled mode to be higher than predicted in the compile time schedule. Although each thread in Coupled mode used an average of 46.5 cycles per device evaluation while each thread in STS mode needed only 25 cycles, the interleaving of threads results in a shorter aggregate runtime for Couple mode (274 cycles) compared to STS mode (505 cycles). Figure 4 of the paper shows that Coupled mode requires about half the number of cycles STS mode requires on average to execute the benchmarks.

Unlimited communication between the individual functional unit clusters is most conducive to extracting parallelism, but incurs high cost. Hence, each register file was limited to 3 ports. This was shown to cause only a 28% increase in runtime over an architecture with unlimited buses and register file write ports.

As a point of interest, the processor implementing this architecture has yet to be run in real time 8 years after this paper was published, because of problems with custom packaging. Also, the real processor has about the same area as mentioned in this paper but only has 3 (instead of 4) clusters, of which only 1 has a floating point unit. This was affirmed by one of the authors, who incidentally was also the discussion leader.

4 *Multiscalar Processors (Sohi, Breach, Vijakumar)*

In a multiscalar model, a program is partitioned into tasks, which are sets of basic blocks. A task is a portion of the static instruction sequence whose execution corresponds to a contiguous region of the dynamic instruction sequence. Tasks are not necessarily control independent whereas threads are. A task is executed only if the dynamic instruction path/stream through the program includes that task; individual threads are executed unconditionally. The multiscalar processor executes multiple tasks in parallel without checking for inter-task control dependencies. Individual tasks are assumed to be control independent and executed accordingly, then checked later and squashed if found to be dependent. (This is similar to the general case presented in the Introduction.)

The rationale of this execution model is that data dependence between tasks with a relatively large number of instructions is less likely, since the instructions in different tasks are further away from each other in the instruction stream.

Consequently, this model relies on speculation. It is imperative that instructions ”appear” to be processed sequentially (even if actually executed out of order) to ensure correctness. However, speculation is applied only at the inter-task level (as opposed to the intra-task level) since instructions within a task are executed in sequential order by the sequential processing units proposed. Therefore, branch prediction is limited to those branches at the end of tasks, which are usually more predictable outer loop branches. (Less predictable inner loop branches are within tasks and hence not executed speculatively.) A loose sequential order of tasks is maintained by organizing the processing units into a circular queue.

To maintain register file coherence, some logic is used to synchronize the production of register values by predecessor tasks with the consumption of these values in successor tasks. For each task, a create mask keeps track of values that the task might produce. At compile time, the create mask of task A is set if task A might write to some register. At run time, the processing unit determines when the register will no longer receive a value then releases the register and forwards its value to other units.

An Address Resolution Buffer (ARB) is used to store speculative updates of the data cache and also to detect violations of memory dependencies and initiate correction of violations when necessary.

5 *Data Speculation Support for a Chip Multiprocessor (Hammond, Willey, Olukotun)*

This paper discusses the implementation of support for thread level speculation on the Hydra chip multiprocessor. With both software and hardware mechanisms, Hydra uses data speculation to partition programs into threads that can execute in parallel without regard for data dependencies.

There are 2 main differences between the Hydra architecture and the Multiscalar architecture. First, Hydra does not speculate where threads are (i.e. control dependencies) whereas Multiscalar does. Multiscalar allows loop level data dependencies to flow through its multiple register sets whereas Hydra forces register updates to go through memory. This is because there is no processor-to-processor communication in the chip multiprocessor.

The speculative mechanism works as follows. Speculation is done on either the thread level or the loop level. In thread level speculation, a processor starts executing a thread. When a subroutine thread is encountered, the first processor continues executing the subroutine thread while the next processor speculatively executes the code after the subroutine thread in parallel. In loop level speculation, subsequent iterations of a loop are executed in parallel by the 4 processors. If a speculation violation is detected, the erroneous thread/iteration and all subsequent threads/iterations are killed. This is necessary because the error may have carried through to later threads/iterations.

When a subroutine fork or loop is encountered, a Register Passing Buffer (RPB) for the thread or iteration must be allocated from the free buffer list. The RPB resides in memory and is necessary to temporarily hold a processor's registers to facilitate processor-to-processor communication during register passing. (The on-chip read and write buses are used for processor-memory traffic only. There are no direct interconnections between the processors.)

Each primary data cache line tag includes read and written bits that are used to detect data dependency violations.

- The read bit for a word in the cache line is set when this processor (the processor associated with this L1 cache) reads from the word. If there is a write to the line containing this word by a less speculative processor (detected on write bus), then a RAW violation has occurred.

- The written bit for a word is set when this processor writes to that word. Subsequent reads from this word by this processor will not set the read bit because this word was generated by this processor. This prevents unnecessary violations.

The main barrier to performance of this architecture is the very high overhead of processor-to-processor communication via memory. Also, the software exceptions used to implement speculative control and the backups due to speculative violations incur the same high overheads. As a result, the performance improvement from data speculation is diminished by the memory penalty.